## 基本概念

### 3.3变量

定义变量时使用var操作符。如果只是声明，但没有赋值则会自动获得undefined；

使用一个从未得到声明的变量会发生ReferenceError报错，但是使用一个得到声明但是并没有得到赋值的变量就不会报错，毕竟自动获得了undefined。

注意，typeof比较特殊，他可以检测一个没有得到声明的不存在的变量，得到的结果是一个undefined字符串，而不会发生报错。

<script>

"use strict";

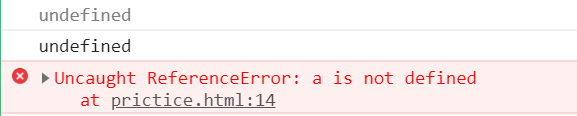
var message;

*console*.log(message); // 就是undefined;

*console*.log(typeof a); // 结果是字符串“undefined”;

*console*.log(a);

</script>



因为var声明的变量的作用域是定义变量下的作用域。因此在脱离这个作用域之后，就会被销毁。最典型的例子就是使用一个函数。

<script>

"use strict";

function a(){ // 这是一个函数作用域

var b = "你好";

}

a(); // b不在全局作用域之中

*console*.log(b); // 报错

</script>



想要保留函数作用域内中的变量，可以去掉var用全局变量的声明，但是这样并不推荐，因为在局部作用域定义的变量很难得到维护，而且也会因为在函数中的全局变量在函数没有运行之前不会得到定义，因此可能会对变成造成不必要的混乱。

在严格模式下对没有经过声明的变量赋值，还会导致ReferenceError错误。

<script>

"use strict";

// var c; 在这里没有声明全局变量c

c = "a";

*console*.log(c); // 抛出错误

</script>



注释掉”use strict”，或者解开var c；的注释就不会发生报错。

在严格模式下也不能定义名为eval或arguments的变量，否则也会导致错误。

### 3.4 数据类型

ECMAScript中定义了五种基本数据类型：

Number、String、Boolean、undefined、null （并没有Object）

Object属于另一种复杂的数据类型。但是不属于基本类型。

所有的值都来自上述6中数据类型之一。

**3.4.1 typeof 运算符**

鉴于ECMAScript是松散类型的，因此需要有能够用来检测三变量类型的方法，可以使用typeof来检测。typeof得到的结果可能是以下几个字符串

‘undefined’ ——如果这个值未定义

‘boolean’ ——如果这个是值布尔值

‘string’——如果这个值是字符串

‘number’——数值

‘object’——对象或者null

‘function’——函数

对于null而言，null在大部分浏览器中会返回“object”，有的浏览器还会返回”function”。因为null在ECMA中被作为空指针，是一个空的对象引用，因此也属于object。

对于函数而言，虽然函数也属于对象，但是因为函数确实有一些特殊的属性。

这里所谓的特殊属性，我认为是函数对象有很多共有的特殊属性，而这些属性不需要定义，一开始就是定义好的，而其他的普通的对象尽管也有共有的属性，但是却不是函数的这些属性。

比如函数属性的length以及arguments.callee.caller等。

详细内容会在之后的高程笔记中详细提到。

3.2.5 Number类型

最基本的是十进制整数，对于是十进制以外的整数，还可以通过八进制或者十六进制的字面值来表示。

但是对于八进制的整数，字面量的第一位必须是0，并且后面的数字不能超过8，否则会忽略前面的0，按照十进制的整数进行解析。

对与十六进制的正式，字面量的前两位必须是0x,否则同八进制的处理方式。

在进行算数计算的时候，最终结果都会转换成十进制数值。

<script *defer*>

var a = 087; // 无效的八进制，解析为87

var c = 011; // 有效的八进制，十进制的9

var b = 0xA; // 十进制的10

*console*.log(c + b); // 十进制的19

</script>

当然也支持科学计数法。

在处理一些数值较大的东西的时候，使用科学计数法是一个不错的选择。

var a = 1e2; // 100

var b = 0.2e-2; // 0.002

var c = 2E3 // 1000

*console*.log(a);

*console*.log(b);

*console*.log(c);

迫于内存的限制，能表示的数值不可能无限大，当然也不可能无限小。

最大值可以通过Number.MAX\_VALUE表示，在大多数浏览器中，这个值一般是5e-324;

最小值可以通过Number.MIN\_VALUE表示，这个值很复杂，就不在这写了，具体还得看浏览器，我使用的是谷歌浏览器做的测试。

*console*.log(*Number*.MAX\_VALUE);

*console*.log(*Number*.MIN\_VALUE);



*console*.log(typeof Infinity); // number

*console*.log(typeof Infinity); // number

*console*.log(*Number*.POSITIVE\_INFINITY); // Infinity

*console*.log(*Number*.NEGATIVE\_INFINITY); // -Infinity



对于这两个正无穷和负无穷，是无法进行数值计算的，如果某次计算结果返回了正或者负的Infinity值，那么该值将无法进行下一次的计算，因为他是不能够参与计算的数值。要想确定一个数值是不是位于最大值和最小值之间，我们可以使用isFinite()函数进行判断，如果在范围之内，就返回true，不在就返回false。

*console*.log(isFinite(1)); // true

*console*.log(isFinite(-Infinity)); // false

虽然说，我们很少会遇见触及最大值或者最小值的值，但是有预防心还是好的。

尽管有点不可思议，但是isNaN或者是其他内置函数和操作符有好多都可以接受一个对象，以isNaN为例，一般流程都是先调用对象的ValueOf方法，然后看他能否转换为数值（我认为应该是，看valueof返回的值能不能判断isNaN的最终结果），如果不能，则基于这个返回值在调用toString()方法（我认为这里不是基于返回值，而是借鉴返回值的情况，来判断是否调用对象的toString方法），在测试返回值。

高程在这个地方说的实在是模糊，而且在之后的说法却不一样，不知道是翻译的问题还是什么，经过我的测试和资料查阅，我觉得我说的是对的。这里放上我的实验代码。

<script *defer*>

var obj = {

valueOf:function(){

*console*.log("走valueof")

return { // 返回的是一个对象，没法转换成数字，要调用外层对象的toString()

valueOf:function(){

*console*.log("走小valueof");

return "2"

},

toString:function(){

*console*.log("走返回值的tostring");

return 1;

}

}

},

toString:function(){

*console*.log("走返回值的toString");

return 3; // 根据上一步结果，这次取toString的返回值，可以转换位数字。

}

};

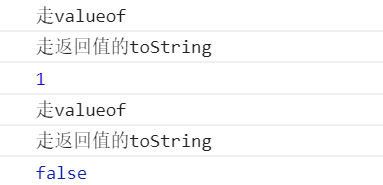
*console*.log(*Number*(obj)); // 3

*console*.log(isNaN(obj)); // false

我的后期补充：需要注意的是，这种先调用valueOf看返回值，不行之后再的方式在调用toString的方式适合大多数注入Number的这种内部函数。

但是在需要明确需要字符串的环境下，比如在alert中，js只会默认调用obj的toString。

而在进行+运算，进行隐式转换的时候，js会盲目的调用valueOf方法，之后不会再尝试调用toString。



<script *defer*>

var obj = {

valueOf:function(){

*console*.log("走valueof")

return "2a" // 不能转换为数字，因此已经可以判断结果为NaN了，所以不用在调用valueof了

},

toString:function(){

*console*.log("走返回值的toString");

return 1;

}

};

*console*.log(*Number*(obj)); // NaN

*console*.log(isNaN(obj)); // true

</script>

